APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/731,060	12/07/2000	Edward Colles Nevill	550-192	1332

23117 7590 12/17/2008
NIXON & VANDERHYE, PC
901 NORTH GLEBE ROAD, 11TH FLOOR
ARLINGTON, VA 22203

EXAMINER
ZHEN, LI B

ART UNIT	PAPER NUMBER
2194	

MAIL DATE	DELIVERY MODE
12/17/2008	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

RECORD OF ORAL HEARING

UNITED STATES PATENT AND TRADEMARK OFFICE

____

BEFORE THE BOARD OF PATENT APPEALS
AND INTERFERENCES

____

Ex parte EDWARD COLLES NEVILL and
ANDREW CHRISTOPHER ROSE

____

Appeal 2008-3073
Application 09/731,060
Technology Center 2100

____

Oral Hearing Held: November 5, 2008

____

Before HOWARD B. BLANKENSHIP, JAY P. LUCAS, and
THU A. DANG, *Administrative Patent Judges.*

.

ON BEHALF OF THE APPELLANT:

John R. Lastova, Esquire
NIXON & VANDERHYE, P.C.
11th Floor
901 North Glebe Road
Arlington, VA 22203

The above-entitled matter came on for hearing on Wednesday, November 5, 2008, commencing at 9:00 a.m., at The U.S. Patent and

Trademark Office, 600 Dulany Street, Alexandria, Virginia, before Dominico Quattrociocchi, Notary Public.

MS. BOBO-ALLEN: Appeal Number 2008-3073. Mr. Lastova.

MR. LASTOVA: Good morning.

JUDGE BLANKENSHIP: Good morning, Mr. Lastova. You have 20 minutes and you can begin whenever you like.

MR. LASTOVA: All right. Well, thank you. What I'd like to do is begin with a little bit of background just to give us all the same context that we're working in. We're discussing a data processing system right now and we are trying to reliably perform scheduling between tasks or threads and multitasking operating systems that are well known, and you have to process resources and share between several different programs that may be simultaneously active.

And one way to control those scheduling operations is a counter-based approach where program instructions are executed and they're counted as they're executed, and a scheduling operation is initiated each time a predetermined program instruction count level is reached. That's, that's one way to do it.

Another way is called sort of a timer-based scheduling approach in which the scheduling operation is initiated at a regular time interval. Sort of like servicing an inner update, they poll the interrupts.

Now we're going to shift to an area where this becomes a little bit trickier, and that is when we go to higher level execution programming language such as Java and Java Virtual Machines which are -- they're, they're platform independent languages, all right.

So what ends up happening is that they run on machines that are designed or architected to execute native instructions which they can do quite quickly. And what they've been able to come up with is they'll say well, we use a, a software interpreter so that you can get these non-native, high-level languages that can be worked across multiple platforms to run there. The problem there is -- of course, is those software interpreters are pretty slow, all right? So they work, but they're slow.

Now the problem comes here that the inventors in this case were concerned with how do we do this scheduling in a multitasking, multithreading environment in this particular context in a way that's both reliable and efficient? And they pointed out that if you use the timer-based approach then you can have a timing that comes out and it says interrupt right now, we want to do the scheduling operation. That can occur in a context switch as you're moving between a hardware-based sort of execution, a native instruction or an interpreted kind of instruction being implemented by a software interpreter, and that can cause some data integrity problems. And the other approach is to use that count value, and what they noticed was as I'm going between a hardware implementation of the simple instructions to a software unit approach for the complicated instructions that's a lot of overhead. I've got to tell each unit that's executing here's the count value, so that's a lot of overhead and that's going to slow things down.

So that's the, that's the context. That's the problem that we're in right now, and I've given you the goal. We want the efficient and the reliable support of both scheduling as well as supporting these high-level languages that aren’t platform dependent.

So if we were to look then at the figures in our case just very briefly, where -- if you look at Figure 1 or Figure 2, we're in the instruction pipeline there and we’re, we’re between the fetch stage and the execution stage. That’s going to give you a big picture view of where we’re at.

And now if you could turn to Figure 10, Figure 10 is the helpful -- I think the most helpful figure to get us focused on how the invention is working. And you can see there’s a hardware side on the left and a software side on the right, and what we have is we have something that we call scheduling logic, which is one example embodiment can be a counter, and you can see that in Step 72 on the hardware execution unit side there’s a decrement counter and a questioning does the counter equal zero?

Well, if it does equal zero, we go ahead and we branch on over to the, to the software side, to some software code that does the scheduling operation, okay. We know that the instruction is finished. We go over, we do the processing, and we’re fine. And then ultimately, after we’ve done the scheduling you see the return feedback loop. We return the control back to the hardware. Then we reset the counter and we move on.

So you can see in Step 78 we’ve fetched the next instruction, which happens to be a Java byte code which is a non-native instruction in this case, and we determine is it a simple byte code, because if it’s a simple byte code then we can simply transmit it. We can quickly put it through the hardware unit, which is basically the translator which says okay here’s the Java byte code and I'm going to output a native instruction that it's essentially the equivalent of that we've done, okay?

So we execute in hardware. As you see in Step 82, we again come back to decrement the counter because we just finished that instruction. We see if it's zero and so forth. Come down to Step 80 again. If it's not a simple byte code -- it's one of those ones we can't translate real quickly into the corresponding native instruction, then we pass the control over to the software handler, or sometimes referred to as a software interpreter, to execute that. And in this case, the software unit is going then have to say okay, well, for this complicated byte code I'm going to have get a series of updated instructions to implement this, and they do that. And while they're done doing that, it comes back around and the control returns to the hardware unit. Once again decrement the counter. Go through the same operation.

And the main point here and the point that we emphasize in Claim 1 especially, elements small Roman Numeral IV and small Roman Numeral V are -- in Roman Numeral IV that -- it says there that the hardware-based execution unit, okay, for which execution is not supported by said hardware based-execution unit are forwarded, that is, the program instructions aren't supported, are forwarded to the software-based execution unit because of the complicated byte codes for execution by that software unit, with control being returned to said hardware-based execution unit for said next program instruction to be executed. So the hardware unit takes control and holds it.

Then in Roman Numeral V, we get to this sort of the scheduling part. The hardware unit includes scheduling support logic. That's that counter in the preferred embodiment -- one of the example embodiments, but there's another embodiment as well to generate a scheduling signal for triggering a scheduling operation to be performed between program instructions, rather than during program instruction, and which would be the problem with the data integrity which we talked about before for managing scheduling between threads or tasks, irrespective of whether proceeding program instruction was executed by the hardware-based execution unit or said software-based execution unit. So the crux here is that we return the control and we route all the program instructions to be executed through the hardware unit. He's our controller, and it keeps track of the execution of the instructions, generates a scheduling signal for triggering a scheduling operation, irrespective of whether that preceding -- immediately preceding instruction was executed by the hardware unit or the software unit.

So the Primary Reference here that we're, we're confronted with is the Evoy reference, and Evoy -- if you look at Figure 2, which perhaps is the most instructive figure that we could look at here in the short time we have, Evoy has a hardware translation unit 50. You kind of see it. It's a little messy there, but right in the middle, if you were to draw a little circle, the hardware translation unit 50 includes that 1 of 4 byte multiplexer 56, that multiplexer right in the middle there, 58, and then this look-up table, 51. And essentially, what that does is, like our translator, brings in the simple byte code. Boom, looks it up in the object table. Out comes the corresponding native instruction and gets routed to that processor, 40, on the right there. And that's fairly straight forward.

Then what ends up happening is if we can't do that, like in our case -- if you can't do that, Evoy has this exceptions signal. You can see the J except which a line -- control signal line, 54-B, which comes out of the object table, 51, goes to the processor. That's the exception, and it puts the processor system in back into the native mode, okay?

And we're going to get into this discussion of native mode and non-native mode because it's very important to try to keep that clear as you're going through Evoy and it's easy to get turned around. I think the Examiner got turned around on this a little bit.

So as you're reading through the reference in Evoy and you're looking at the figure and you're looking at our claims, try to be clear where Evoy is; whether Evoy is in the native mode or the non-native mode, okay, when he's doing that.

When you move to the software interpreter -- when you get to that complicated byte code, what ends up happening is that the system moves to the non-native mode, all right, and it goes ahead and it executes multiple native instructions so that is can essentially implement that complicated byte code using the software interpreter.

Let me give you just a very simple scenario so you can see the difference. I fetch the first byte code in Evoy. It happens to be a simple one. We'll call it byte code 1. It's translated in the hardware unit 50. We're doing fine. We're in the non-native mode. We've translated. Boom, it's gone. It's executed. The address counter gets incremented by one. Fetch the next byte code, okay. We'll call that byte code 2, but that happens to be complicated. That's a complicated byte code now. I can't translate anymore in the non-native mode using this translator. I've got to go exception switch to native mode now. I've got my software interpreter, and he says okay I, I see the complicated instruction. I'm in the native mode. I've got three native instructions, 1, 2, and 3 that I'm going to use to implement that, that complicated byte code, all right?

So I go ahead, I implement and I execute native instruction number 1.

JUDGE LUCAS: Excuse me. The software. I'm sorry, I --

MR. LASTOVA: Yeah.

JUDGE LUCAS: Mr. --

MR. LASTOVA: Lastova.

JUDGE LUCAS: -- Lastova. You say you have the three native byte codes.

MR. LASTOVA: No, no. So, so complicated byte code number 2 --

JUDGE LUCAS: Right, is now being interpreted by the software interpreter?

MR. LASTOVA: Interpreter --

JUDGE LUCAS: Yes.

MR. LASTOVA: -- and the software interpreter says to implement that complicated byte code number 2. I'm going to use three native instructions.

JUDGE LUCAS: Yes.

MR. LASTOVA: Okay. So he implements native instruction 1, executes that. Good to go, all right. Where does the control go? Doesn't go back to the hardware unit to the translator; stays with the interpreter because I've got to do --

JUDGE LUCAS: You have to do all three. Right?

MR. LASTOVA: I have to do all three. So that's where it stays. Control does not go back to the translation unit -- the hardware unit here.

JUDGE LUCAS: While it is processing that one line of complicated Java byte code?

MR. LASTOVA: That's right, that's right, but it's an instruction. Right? Each one of those three instructions I just told you about those data instructions is an instruction, right? Each one of those is being executed, right? It's a program instruction, right?

JUDGE LUCAS: The higher level byte code instruction is also an instruction.

MR. LASTOVA: That's true.

JUDGE LUCAS: Okay.

MR. LASTOVA: We could see it that way.

JUDGE LUCAS: So it's processing that higher level byte code with three steps, as indicated by the software interpreter in your example?

MR. LASTOVA: Right, three, three native instructions.

JUDGE LUCAS: Three native instructions.

MR. LASTOVA: All right.

JUDGE LUCAS: Okay, I understand your position. Continue on.

MR. LASTOVA: Sure. So what I'm saying here is that Element 4 in Claim 1 is missing. The Examiner relies on Evoy and he just says well, all I need to do is to find basically an incrementing grabbing the next byte code and I'm done. And what I'm pointing out here is that the claim requires the control to be returned to the hardware execution for our next program instruction to be executed. And we just said in my little scenario the next program instruction to be interpreted -- I mean to be executed is the native instruction number 2.

I haven't finished executing with the software interpreter yet that complicated byte code, and so what ends up happening here is that control never gets back to the hardware unit. And the important point here is -- Judge Lucas, you had some questions on this. I just want to direct this to, to your attention --

JUDGE LUCAS: I'm listening to you.

MR. LASTOVA: Okay. Is that there is going to be a problem, and I can see there would be some nuance once the program instruction, but here's the interesting thing. If in Evoy you were to have some kind of interrupt occur, there has to be a scheduling operation, and it occurs during the implementation of those three native instructions. And right after you finish native instruction number 1, okay -- I have to go off to do the scheduling operation -- you can have some data integrity problems by the time I get back and have to complete native instruction 2 and 3 to get the full execution.

So the problem here is not avoided by sort of some gaming of what does a "program instruction" mean, and that's a point I just want to make real quick.

Now what's the Examiner's position on this? The Examiner is just saying well, you know, I'm going to point to the -- select the next byte code in Column 7 and Lines 8 through 15. We don't have time to read all the text, but what I would say is this: Read the text carefully, starting from the bottom of Column 6 all the way through Column 7, and watch very carefully what mode you're in, okay? Because if you're selecting the next byte code, you're in the non-native mode. You're not in the software interpreter's control anymore. You're still with the hardware unit; you're non-native mode.

And when you're selecting the next byte code, all you're doing is just rattling around with these simple byte codes and saying, oh, simple byte codes, simple byte code. We just stay with the hardware unit, and that's fine. The problem is what happens in Evoy when you get the complicated byte code? And that's the point I was trying to give you that little scenario for. It breaks down, one. Evoy is a little bit fuzzy themselves and don't go into elaborate discussion here, but if you follow through and try to see where the modes are, you'll see that the software interpreter retains control throughout the execution of those native instructions that it is doing to implement the complicated byte code.

So this, this fundamental misunderstanding --

JUDGE LUCAS: Excuse me.

MR. LASTOVA: Yes, sir.

JUDGE LUCAS: In your reading of Evoy, after the software interpreter finished its job --

MR. LASTOVA: Right.

JUDGE LUCAS: -- of interpreting that complicated line byte code, then is the program counter moved on and it goes to the next line of code?

MR. LASTOVA: The program counter will go to -- will be incremented, that's true. The program counter will have to go to a next instruction so that the decoder --

JUDGE LUCAS: I'm talking --

MR. LASTOVA: -- can fetch the next instruction.

JUDGE LUCAS: -- after all three are done.

MR. LASTOVA: But -- I'm sorry>

JUDGE LUCAS: After all three of the complicated lines of byte code are translated into the three instructions --

MR. LASTOVA: Native instructions.

JUDGE LUCAS: Native instructions. At that point, will the Evoy system then move on to the next line of instruction?

MR. LASTOVA: It will move on to a next line of instruction. The question to be asking is what mode is it in and who has control.

JUDGE LUCAS: That's a question point.

MR. LASTOVA: Yep. All right, so that's a fundamental misunderstanding and one in which I think where the Examiner has glossed over and perhaps --

JUDGE BLANKENSHIP: Which has control in the reference?

MR. LASTOVA: In the Evoy reference --

JUDGE BLANKENSHIP: Right.

MR. LASTOVA: -- I can say it's not clear. They don't point out what it is. All you can really get from Evoy is that the decoder 45 as you'll see in Figure 5, depending on what is fetched, determines the next mode you go in. So if the next instruction that is fetched happens to be a complicated

byte code, again, you're still in the software interpreter having control, right?
In other words, if I can't -- if the next thing I, I fetch from memory -- my next instruction incremented I fetch is a complicated byte code --

JUDGE BLANKENSHIP: I understand.

MR. LASTOVA: -- it's got to go back to the software interpreter. Never goes back to the hardware again. Does that answer your question?

JUDGE BLANKENSHIP: Well, so sometimes it sounds like it would go -- in that interpretation it sounds like sometimes it would go back to hardware control.

MR. LASTOVA: It would, but, as I was telling Judge Lucas, it still has to stay with the software interpreter -- the execution unit for the three native instructions that come consecutively to implement the first complicated byte code. So either way we're going to have trouble with, with the Examiner's application of Evoy.

In both of those scenarios that we just talked about, you're having problems because the control does not get back to the hardware unit. If the next instruction happens to be a simple byte code, you're, you're correct, you come over to the hardware unit and say okay, now we're in the non-native mode. This is a simple byte code, go ahead and translate it. But my point here to Judge Lucas was well, when we were doing the complicated byte code and we translate -- we interpreted it into three native instructions -- software interpreter -- software unit executes the first one, second one, third one. Control stays with the software unit after each execution of those three. So I think that's the main point I'm trying to make right here.

Okay so that's one -- I think grounds for reversal right there because the Examiner is not relying on Gee for that particular feature. The second main thing I'd like to talk today, and I've got a few more minutes remaining, is Element 5. Element 5 which I read to you earlier on, and I won't repeat it here, the Examiner is basically relying on the Gee reference to, to supply that missing feature. And Gee is -- basically what he's relying on is primarily Column 21, and what he's pointing out there is that in conventional Java scheduling type systems and even refers to a particular section and that's how conventional it is. It implements a preemptive priority-based scheduling policy that just says if you look at the rules there are three simple rules. The main rule that's of interest here is that it says if a blocked higher priority thread becomes runnable during the execution of a lower priority thread, the lower priority thread, that is preempted and the higher priority thread executed.

JUDGE LUCAS: What about Rule 1?

MR. LASTOVA: It says -- well, it says the currently executed thread is always the highest priority runnable thread.

JUDGE LUCAS: Therefore, if it's executing a thread it won't interrupt the thread that it's actually working on?

MR. LASTOVA: But if it's blocked -- I'm reading 1 and 2 together, so to me, as I look at those two, they have to be read together. So he's saying look, the currently it's the highest priority unless there was a blocked higher priority thread. If it becomes runnable during the execution of a lower priority thread, even though it may be currently running, the lower priority thread is preempted and the higher priority thread executed. But anyway, I mean --

JUDGE LUCAS: That's Rule 2.

MR. LASTOVA: That's Rule 2, but those are rules that, that you can't just look at one rule. You have to look at all three rules together, right.

So anyway my, my main point here is just to point out that when you look at Element 5 don't disconnect Element 5 from Element 4. Read the claim as a whole and in its entirety, all right? And when you look at that -- let's assume we take this conventional priority-based scheduling from G and put into Evoy, all right, just for purposes of discussion here, okay? What we see is you still lack this hardware unit keeping control, all right, after the executions of each program instruction. And at the same time, all right, the hardware unit generating the scheduling signal, all right, to be performed when program instructions for managing scheduling between threads and tasks, irrespective of whether a preceding program instruction was executed by said hardware-based execution unit or said software-based executing unit.

In other words, come back to that idea that it's got to come back to the hardware unit so the hardware unit can control the execution of the next instruction and, as we saw in Figure 10, in that example, can send out the scheduling instruction to the scheduler at the appropriate point in time, not in the execution during -- of an ongoing instruction, which is my concern about point number 2. But even if we just ignore that for a moment, the bottom line is this priority-based scheduling doesn't get to this control being returned to the hardware unit, irrespective of whether the last instruction was executed of the hardware-based unit or the software-based execution in it.

There is a number -- I can see my time is up, so what I would say is this I would just refer you to the Brief for obviously some of the other arguments that we talk about, and I have -- I think there's -- it's important that if you're concerned about the scenario I just gave you, Judge Lucas, you would read Column 5, Lines 64 through 67 and Column 6, Line 4 through 8 --

JUDGE LUCAS: Of ?

MR. LASTOVA: Of Evoy, just to get that sense of what's happening in the, in the native mode when a software interpreter is, is going there. So I, I saved just a couple of minutes here. I don't know how much time I have, but -- for any other questions.

JUDGE LUCAS: I think you've given a very thorough explanation.

MR. LASTOVA: Thank you.

JUDGE LUCAS: And I thank you very much.

MR. LASTOVA: Thank you.

JUDGE DANG: Thank you.

MR. LASTOVA: Have a good day.

(Whereupon, the hearing concluded on November 5, 2008.)